What this lecture is about: Secrets of number encoding, how to hack integers (and floats). 😊
Today

- Representation of numbers – introduction.
- Integer coding.
- Basic arithmetic.
- Hexadecimal notations.
- Endianness.
- IEEE floats.
- Implementation of functions.
- Numerical precision.
Basics

Natural/Real Numbers
- Base 10
- Infinite
- Exact

Computer Numbers
- Base 2
- Finite
- Rounding - overflow

In this lecture
- How to represent numbers – range, encoding.
- Arithmetic.
- How to use these numbers.

Unit for coding = bit.
Finite number of bits, finite numbers.
Signed/unsigned: similar representations but different interpretations.
Questions

- How to code negative numbers?
- How to code real numbers?
- Which kind of precision do we get?
  - Small numbers vs. big numbers.
Example

- Overflow:
  ```c
  main() {
      printf("%d\n", 200*300*400*500);
  }
  ```
  outputs -88490188.

- Fix – sort of:
  ```c
  main() {
      printf("%lld\n", 200LL*300LL*400LL*500LL);
  }
  ```
  outputs 12000000000.

What if I forget LL?

Limits of int32_t: \(-2^{31}…2^{31}-1\).
Example

- Loss of precision:
  \[(3.14+1e20)-1e20==0.0\]
  \[3.14+(1e20-1e20)==3.14\]

- Test \(x == 0.0\) not very useful when solving equations.
Information Storage

- Basic unit is the byte (=8 bits).

<table>
<thead>
<tr>
<th>C-declaration</th>
<th>Typical 32-bit</th>
<th>Compaq Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Size of the different types of integers depends on the architecture. Addressing is limited by the size of pointers that gives the size of addressable memory.
Features

- Limits on addressable memory.
- Size of registers – 32/64.
- Aligned memory allocation (32/64 bits).
- Careful on addressing:
  ```c
  main() {
    char a[]="Hello world!";  // ?
    int *p=&a[1];
    printf("%d\n",*p);
  }
  ```
**Integer Coding**

- **Unsigned integers:** \( UB = \sum_{i=0}^{w-1} x_i 2^i \)

- **Signed integers:** \( SB = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \)
  
  Called 2 complement.

- Highest bit codes the sign.

\( w: \) size of a word (in bits)
\( x: \) bits (0 or 1)
Basic Arithmetic

- Logical operations (bitwise):
  &, |, ^, ~, <<, >>.
  Example: a ^= b; b ^= a; a ^= b;

- Arithmetic operations: + - * /.

- Careful with shifts on signed integers!

- Do not mess up with boolean operations (&&, ||).
Properties

- Operations are the same on signed/unsigned integers.
- Check properties on the notes.
- Operations based on the algebra $\langle \mathbb{Z}_n, +_n, \cdot_n, -_n, 0, 1 \rangle$. Operations modulo $n$ and $-a=0$ or $-a=n-a$.

Basic properties:
- commutativity: $a+b=b+a$, $a*b=b*a$
- associativity: $(a+b)+c=a+(b+c)$, $(a*b)*c=a*(b*c)$
- distributivity: $a*(b+c)=a*b+b*c$
- identities: $a+0=a$, $a*1=a$
- annihilator: $a*0=0$
- cancellation: $(-a)=a$

Properties for integer ring $\langle \mathbb{Z}, +, \cdot, -, 0, 1 \rangle$ and for boolean algebra $\langle \{0,1\}, |, &, \neg, 0, 1 \rangle$ are similar (| instead of +, & instead of *).

Unique for integer ring: $a+-a=0$.
Unique for boolean algebra:
- distributivity: $a|(b\&c) = a\&(b|c)$
- complement: $a|\neg a=1$, $a\&\neg a=0$
- idempotency: $a\&a=0$, $a|a=a$
- absorption: $a|(a\&b)=a$, $a\&(a|b)=a$
- DeMorgan laws: $\neg(a\&b)=\neg a\neg b$, $\neg(a\|b)=\neg a\neg b$
Shifts & Masks

- Read bit n: Use mask (1 << n).
- Set bit n on int bits[]:
  \[ \text{ipos} = \frac{n}{32}; \]
  \[ \text{imask} = 1 << (n \mod 32); \]
  \[ \text{bits[ipos]} |= \text{imask}; \]
- Shifts as division/multiplication by powers of 2.
- Shifts for negative numbers has “wrong” rounding.
  Correct: \((x<0?(x+(1<<k)-1):x)>>k\).
How does it work?

```
1011
+1101
-----
111
11000
```

Subtraction?

```
1011
*1101
-----
1011
0000
1011
1011
-----
10001111
```

**multiplicand**

**multiplier**

**partial products**

**product**
How does it work?

\[ \frac{10010011}{1011} = ? \]

\[ \begin{array}{c|c}
\text{divisor} & 1011 \\
\hline
\text{dividend} & 10010011 \\
\text{quotient} & \end{array} \]

\[ \begin{array}{r}
0 \\
1011 \overline{10010011} \\
1011 \\
00 \\
1011 \overline{10010011} \\
1011 \\
000 \\
1011 \overline{10010011} \\
1011 \\
00001 \\
1011 \overline{10010011} \\
1011 \\
000011 \\
1011 \overline{10010011} \\
1011 \\
001110 \\
1011 \\
\end{array} \]

partial remainder
Block Diagram for A+B, A-B

B Register

2-complementer

switch +/-

overflow

A Register

Adder

overflow
Block Diagram for M*Q

Multiplicand

\[ M_{n-1} \ldots M_0 \]

n-bit adder

\[ C \rightarrow A_{n-1} \ldots A_0 \rightarrow Q_{n-1} \ldots Q_0 \]

Multiplier

\[ Q_{n-1} \ldots Q_0 \]

Shift & add logic

\[ A_{n-1} \ldots A_0 \rightarrow C \rightarrow \text{add} \]

Shift right

Product in A,Q
Flowchart for unsigned *

C,A ← 0
M ← multiplicand
Q ← multiplier
count ← n

Q₀ = 1?

yes

C,A ← A+M

no

right shift C,A,Q
count ← count-1

count = 0?

no

yes

product in A,Q

Product in A,Q
Flowchart for unsigned /

A ← 0
M ← divisor
Q ← dividend
count ← n

shift left A, Q
A ← A - M

A < 0?

Q₀ ← 1

no

A ← A - M

count ← count - 1

yes

Q₀ ← 0
A ← A + M

count = 0?

no

yes

quotient in Q
remainder in A
Arithmetic

- Machine code of + - * / same for int/uint.
- Integer conversion == type casting.
  - Padding for the sign (int).
  - Conversion is modulo the size of the new int.
  - Beware of implicit conversions in C!
- Optimizations for some operations:

  \[ 2^a \equiv a+a \equiv a\ll1 \quad a/2 \equiv a\gg1 \]
  \[ a\times2^i \equiv x \ll i \quad a/2^i \equiv a\gg i \]
  \[ a\%2^i \equiv a \& (1\ll i) - 1 \quad 2^i \equiv 1\ll i \]

Previous read example for a positive index \( n \):
\[
\text{ipos} = n \gg 5;
\]
\[
\text{imask} = 1 \ll (n \& 31);
\]
\[
\text{bits}[n] |\neq \text{imask};
\]
Notes

- Beware of precedence of operators:
  - if (x & mask == value) WRONG
  - if ((x & mask) = value) RIGHT
- Test odd numbers: if (x & 1)
- Careful:
  unsigned int i;
  for(i = 0; i < n-1; ++i) ...
Hexadecimal Notation

- Learn the first powers of 2.
- Hexadecimal more useful:
  - One digit codes 4 bits. 0...F=0...15=16 numbers.
  - C notation: 0x...
- Examples:
  - $0xa57e = 1010\ 0101\ 0111\ 1110$
    $10=8+2,5=4+1,7=4+2+1,14=8+4+2$
  - $0xf = 1111,\ 0x7=0111,\ 0x3=0011$

Remember: Individual bits are accessed by shifts and masks.
Ranges:
- uint: 0...0xffffffff
- int: 0x80000000...0x7fffffff

0xf, 0x7, 0x3, 0x7f, ... are strings of consecutive 1s.
To get the encoding of negative numbers, use -a=-a+1.
Allocated memory is 32 bits aligned
- 2 lower bits are = 0.
- Use them to store data.
- But it’s a hack.
Endianness: Beware!

- “0xa57e” is a notation for humans. Corresponds to “1010 0101 0111 1110” in base 2.
  - Little endian: stored as 011111010100101.
  - Big endian: stored as 1010010101111110.
- Does not matter in C, except for
  - bitmap manipulation
  - device drivers
  - network transfers
Testing for endianness

- Write a value on 32 bits.
  - Read 8/16 bits and check what was written.
  - Exercise for Sun/Intel.

```c
main() {
    int a = 0xf0000000;
    char *c = &a;
    printf("%x\n", *c);
}
```

- Intel? 0
- Sun? ffffffff0
- What if a = 0x70000000?

• endianness
• sign coding
• integer conversion
Representation of reals

- How to code a real number with bits?
  - Finite precision → approximation.
  - Represent very small and very large numbers → density of encoding varies.
- Scientific notation used, e.g. (base 10), 3.141e2 – but in base 2.
- Starter: fractional numbers – bad for large or small numbers.
  - Decimal (d):
  - Binary (b):
    \[ d = \sum_{i=-n}^{m} 10^i d_i \]
    \[ b = \sum_{i=-n}^{m} 2^i b_i \]
IEEE floats

IEEE floating point standard

- \( V = (-1)^s M \times 2^E \)
- Number of bits (float/double):
  - \( s[1], m[23/52], e[8/11] \).
- Normalized and de-normalized values.
- Bit fields: \( s, m, e \) to code respectively \( S, M, E \).
- Normalized values (\( e \neq 0, e \neq 111 \ldots \))
  - \( E = e - \text{bias} \) (-126...127/-1022...1023).
  - \( M = 1 + m \) \((1 \leq M < 2)\)
  - **Trick for more precision: implied leading 1**

Bias is used for a smooth transition between normalized and de-normalized numbers. IEEE distinguishes between +0.0 and -0.0: one more reason to test with a tolerance.

NaN generated for \(\sqrt{-1}, \text{inf-inf}, 0/0\).
IEEE floats

- De-normalized values (e= 0 or 11...)
  - e=0: E=1-bias, bias=\(2^{k-1}-1\)
    - Coding compensates for M not having an implied leading 1.
    - M=m
    - For numbers very close to 0.
  - e=11...:
    - m=0, (signed infinite)
    - m!=0, NaN.
Features of IEEE floats

- +0.0 == 0 (binary representation = 00...).
- If interpreted as unsigned int, floats can be sorted (+x ascending, -x descending).
- All int values representable by doubles.
- Not all int values representable by floats.
  - round to even (avoid stat. bias)
  - round towards 0
  - round up
  - round down
  - can’t choose in C...
Properties

- Operations NOT associative.
- Not always inverse (infinity).
- Loss of precision.
- Ex: \( x = a + b + c; \ y = b + c + d; \) Optimize or not?
- Monotonicity \( a \geq b \Rightarrow a + x \geq b + x \)
- Casts:
  - int2float rounded, double2float rounded/overflow
  - int2double, float2double OK
  - float2int, double2int truncated/rounded/overflow.

Important for compilers and programmers.
IA32: The good and the bad

- Good: Uses internally 80 bits extended registers for more precision.
- Bad:
  - Stack based.
  - Side effects like changing values when loading or saving numbers in memory whereas register transfers are exact.
- Extensions: MMX, SSE, Altivec. SIMD instructions = operations working in parallel on multiple data.
Implementation - summary

Integers
- addition/subtraction simple
- multiplication:
  r=0; while(b) do { if (b&1) r+=a; a+=a; b>>=1; }
- division iterative

Floats
- addition/subtraction complicated
  - check for 0, align significands, add/sub, normalize
  - guard bits to avoid losing precision on very close numbers
    \((1.00\ldots2^1 - 1.11\ldots2^0)\)
- multiplication/division principle simpler
  - multiply/divide significands, add/sub exponents, detect
    over/under-flow.
Complex functions

- Lagrange polynomials
- Taylor series

\[ P_n(x) = \sum_{i=0}^{n} f(x_i) \prod_{k=0, k \neq i}^{n} \frac{x - x_k}{x_i - x_k} \]

\[ f(x) = \sum_{i=0}^{+\infty} f^{(i)}(x_0) \frac{(x - x_0)^i}{i!} \]

- Other numerical methods that converge rapidly.
- Intel:: cos, sin, sqrt, in hardware.
- Special (int): random generator \( x_{i+1} = (ax_i + c) \mod m \).
  - Simple but correlation between successive values.
  - Higher bits better quality than lower bits.
Numerical precision

- Evaluation of precision
  - absolute $x \pm \alpha$
  - relative $x^*(1 \pm \alpha)$

- Be careful with division by very small values: Can amplify numerical errors.
  - Numerical justification for Gauss’ method to solve linear equations.

- More in courses on numerical methods.